About size_t and ptrdiff_t

Andrey Karpov
OOO "Program Verification Systems"

September 2009

Abstract

The article will help the readers understand what size_t and ptrdiff_t types are, what they are used for and when they must be used. The article will be interesting for those developers who begin creation of 64-bit applications where use of size_t and ptrdiff_t types provides high performance, possibility to operate large data sizes and portability between different platforms.

Introduction

Before we begin I would like to notice that the definitions and recommendations given in the article refer to the most popular architectures for the moment (IA-32, Intel 64, IA-64) and may not fully apply to some exotic architectures.

The types size_t and ptrdiff_t were created to perform correct address arithmetic. It had been assumed for a long time that the size of int coincides with the size of a computer word (microprocessor's capacity) and it can be used as indexes to store sizes of objects or pointers. Correspondingly, address arithmetic was built with the use of int and unsigned types as well. int type is used in most training materials on programming in C and C++ in the loops' bodies and as indexes. The following example is nearly a canon:

```
for (int i = 0; i < n; i++)
  a[i] = 0;
```

As microprocessors developed over time and their capacity increased, it became irrational to further increase int type's sizes. There are a lot of reasons for that: economy of memory used, maximum portability etc. As a result, several data model appeared declaring the relations of C/C++ base types. Table N1 shows the main data models and lists the most popular systems using them.

short	int	long	ptr	long long	Label		Examples
...	16	...	16	...	IP16		PDP-11 Unix (1973)
16	16	32	16	...	IP16L32		PDP-11 Unix (1977); multiple instructions for long
16	16	32	32	...	I16LP32		MC68000 (1982); Apply Macintosh 68K; Microsoft operation systems (plus extras for x86 segments)
16	32	32	32	...	ILP32		IBM 370; VAX Unix; many workstation
16	32	32	32	64	ILP32LL or ILP32LL64		**Microsoft Win32;** Amdahl; Convex; 1990 Unix systems; Like IP16L32, for same reason; multiple instructions for long long
16	32	32	64	64	LLP64 or IL32LLP64 or P64	64-bit systems	**Microsoft Win64** (X64 / IA64)
16	32	64	64	64	LP64 or I32LP64		**Most Unix systems** (Linux, Solaris, DEC OSF/1 Alpha, SGI Irix, HP UX 11)
16	64	64	64	64	ILP64		HAL; logical analog of ILP32
64	64	64	64	64	SILP64		UNICOS

Table N1. Data models

As you can see from the table, it is not so easy to choose a variable's type to store a pointer or an object's size. To find the smartest solution of this problem size _t and ptrdiff_t types were created. They are guaranteed to be used for address arithmetic. And now the following code must become a canon:

```
for (ptrdiff_t i = 0; i < n; i++)
  a[i] = 0;
```

It is this code that can provide safety, portability and good performance. The rest of the article explains why.

size_t type

size_t type is a base unsigned integer type of C/C++ language. It is the type of the result returned by sizeof operator. The type's size is chosen so that it could store the maximum size of a theoretically possible array of any type. On a 32-bit system size_t will take 32 bits, on a 64-bit one 64 bits. In other words, a variable of size_t type can safely store a pointer. The exception is pointers to class functions but this is a special case. Although size_t can store a pointer, it is better to use another unsinged integer type uintptr_t for that purpose (its name reflects its capability). The types size_t and uintptr_t are synonyms. size_t type is usually used for loop counters, array indexing and address arithmetic.

The maximum possible value of size_t type is constant SIZE_MAX.

ptrdiff_t type

ptrdiff_t type is a base signed integer type of C/C++ language. The type's size is chosen so that it could store the maximum size of a theoretically possible array of any type. On a 32-bit system ptrdiff_t will take 32 bits, on a 64-bit one 64 bits. Like in size_t, ptrdiff_t can safely store a pointer except for a pointer to a class function. Also, ptrdiff_t is the type of the result of an expression where one pointer is subtracted from the other (ptr1-ptr2). ptrdiff_t type is usually used for loop counters, array indexing, size storage and address arithmetic. ptrdiff_t type has its synonym intptr_t whose name indicates more clearly that it can store a pointer.

Portability of size_t and ptrdiff_t

The types size_t and ptrdiff_t enable you to write well-portable code. The code created with the use of

size_t and ptrdiff_t types is easy-portable. The size of size_t and ptrdiff_t always coincide with the pointer's size. Because of this, it is these types that should be used as indexes for large arrays, for storage of pointers and pointer arithmetic.

Linux-application developers often use long type for these purposes. Within the framework of 32-bit and 64-bit data models accepted in Linux, this really works. long type's size coincides with the pointer's size. But this code is incompatible with Windows data model and, consequently, you cannot consider it easy-portable. A more correct solution is to use types size_t and ptrdiff_t.

As an alternative to size_t and ptrdiff_t, Windows-developers can use types DWORD_PTR, SIZE_T, SSIZE_T etc. But still it is desirable to confine to size_t and ptrdiff_t types.

Safety of ptrdiff_t and size_t types in address arithmetic

Address arithmetic issues have been occurring very frequently since the beginning of adaptation of 64-bit systems. Most problems of porting 32-bit applications to 64-bit systems relate to the use of such types as int and long which are unsuitable for working with pointers and type arrays. The problems of porting applications to 64-bit systems are not limited by this, but most errors relate to address arithmetic and operation with indexes.

Here is a simplest example:

```
size_t n = ...;
for (unsigned i = 0; i < n; i++)
  a[i] = 0;
```

If we deal with the array consisting of more than UINT_MAX items, this code is incorrect. It is not easy to detect an error and predict the behavior of this code. The debug-version will hung but hardly will anyone process gigabytes of data in the debug-version. And the release-version, depending on the optimization settings and code's peculiarities, can either hung or suddenly fill all the array cells correctly producing thus an illusion of correct operation. As a result, there appear floating errors in the program occurring and vanishing with a subtlest change of the code. To learn more about such phantom errors and their dangerous consequences see the article "A 64-bit horse that can count" [1].

Another example of one more "sleeping" error which occurs at a particular combination of the input data (values of A and B variable):

```
int A = -2;
unsigned B = 1;
int array[5] = { 1, 2, 3, 4, 5 };
int *ptr = array + 3;
ptr = ptr + (A + B); //Error
printf("%i\n", *ptr);
```

This code will be correctly performed in the 32-bit version and print number "3". After compilation in 64-bit mode there will be a fail when executing the code. Let's examine the sequence of code execution and the cause of the error:

- A variable of int type is cast into unsigned type;
- A and B are summed. As a result, we get 0xFFFFFFFF value of unsigned type;
- "ptr + 0xFFFFFFFFu" expression is calculated. The result depends on the pointer's size on the current platform. In the 32-bit program, the expression will be equal to "ptr - 1" and we will successfully print number 3. In the 64-bit program, 0xFFFFFFFFu value will be added to the pointer and as a result, the pointer will be far beyond the array's limits.

Such errors can be easily avoided by using size_t or ptrdiff_t types. In the first case, if the type of "i" variable is size_t, there will be no infinite loop. In the second case, if we use size_t or ptrdiff_t types for "A" and "B" variable, we will correctly print number "3".

Let's formulate a guideline: wherever you deal with pointers or arrays you should use size_t and ptrdiff_t types.

To learn more about the errors you can avoid by using size_t and ptrdiff_t types, see the following articles:

- 20 issues of porting C++ code on the 64-bit platform [2];
- Safety of 64-bit code [3];
- Traps detection during migration of C and C++ code to 64-bit Windows [4].

Performance of code using ptrdiff_t and size_t

Besides code safety, the use of ptrdiff_t and size_t types in address arithmetic can give you an additional gain of performance. For example, using int type as an index, the former's capacity being different from that of the pointer, will lead to that the binary code will contain additional data conversion commands. We speak about 64-bit code where pointers' size is 64 bits and int type's size remains 32 bits.

It is a difficult task to give a brief example of size_t type's advantage over unsigned type. To be objective we should use the compiler's optimizing abilities. And the two variants of the optimized code frequently become too different to show this very difference. We managed to create something like a simple example only with a sixth try. And still the example is not ideal because it demonstrates not those unnecessary data type conversions we spoke above, but that the compiler can build a more efficient code when using size_t type. Let's consider a program code arranging an array's items in the inverse order:

```
unsigned arraySize;
...
for (unsigned i = 0; i < arraySize / 2; i++)
{
  float value = array[i];
  array[i] = array[arraySize - i - 1];
  array[arraySize - i - 1] = value;
}
```

In the example, "arraySize" and "i" variables have unsigned type. This type can be easily replaced with size_t type, and now compare a small fragment of assembler code shown on Figure 1.

array[arraySize - i - 1] = value;	
arraySize, i : unsigned	arraySize, i : size_t

mov eax, DWORD PTR arraySize$[rsp]
sub eax, r11d
add r11d, 1
add eax, -1
movss DWORD PTR [rbp+rax*4], xmm0
...
	mov rax, QWORD PTR arraySize$[rsp]
	sub rax, r11
	add r11, 1
	
	movss DWORD PTR [rdi+rax*4 - 4], xmm0
	...

Figure N1.Comparison of 64-bit assembler code when using unsigned and size_t types

The compiler managed to build a more laconic code when using 64-bit registers. I am not affirming that the code created with the use of unsigned type will operate slower than the code using size_t. It is a very difficult task to compare speeds of code execution on modern processors. But from the example you can

see that when the compiler operates arrays using 64-bit types it can build a shorter and faster code.

Proceeding from my own experience I can say that reasonable replacement of int and unsigned types with ptrdiff_t and size_t can give you an additional performance gain up to 10% on a 64-bit system. You can see an example of speed increase when using ptrdiff_t and size_t types in the fourth section of the article "Development of Resource-intensive Applications in Visual C++" [5].

Code refactoring with the purpose of moving to ptrdiff_t and size_t

As the reader can see, using ptrdiff_t and size_t types gives some advantages for 64-bit programs. However, it is not a good way out to replace all unsigned types with size_t ones. Firstly, it does not guarantee correct operation of a program on a 64-bit system. Secondly, it is most likely that due to this replacement, new errors will appear data format compatibility will be violated and so on. You should not forget that after this replacement the memory size needed for the program will greatly increase as well. And increase of the necessary memory size will slow down the application's work for cache will store fewer objects being dealt with.

Consequently, introduction of ptrdiff_t and size_t types into old code is a task of gradual refactoring demanding a great amount of time. In fact, you should look through the whole code and make the necessary alterations. Actually, this approach is too expensive and inefficient. There are two possible variants:

1. To use specialized tools like Viva64 included into PVS-Studio. Viva64 is a static code analyzer detecting sections where it is reasonable to replace data types for the program to become correct and work efficiently on 64-bit systems. To learn more, see "PVS-Studio Tutorial" [6].
2. If you do not plan to adapt a 32-bit program for 64-bit systems, there is no sense in data types' refactoring. A 32-bit program will not benefit in any way from using ptrdiff_t and size_t types.

References

1. Andrey Karpov. A 64-bit horse that can count. http://www.viva64.com/art-1-2-377673569.html
2. Andrey Karpov, Evgeniy Ryzhkov. 20 issues of porting C++ code on the 64-bit platform. http://www.viva64.com/art-1-2-599168895.html
3. Andrey Karpov. Safety of 64-bit code. http://www.viva64.com/art-1-2-416605136.html
4. Andrey Karpov, Evgeniy Ryzhkov. Traps detection during migration of C and C++ code to 64-bit Windows. http://www.viva64.com/art-1-2-2140958669.html
5. Andrey Karpov, Evgeniy Ryzhkov. Development of Resource-intensive Applications in Visual C++. http://www.viva64.com/art-1-2-2014169752.html
6. Evgeniy Ryzhkov. PVS-Studio Tutorial. http://www.viva64.com/art-4-2-747004748.html